# Estructura backend para una aplicación de comercio electrónico

Estructurar el backend de una aplicación de comercio electrónico requiere una planificación cuidadosa para garantizar escalabilidad, mantenibilidad, seguridad y eficiencia. A continuación, se describe una posible estructura del backend, incluyendo las tecnologías a utilizar, la organización de los archivos y los patrones de diseño recomendados.

## Tecnologías

Optamos por Node.js para implementar el backend de nuestro sistema de comercio electrónico, para aprovechar su naturaleza de E/S sin bloqueo y su ecosistema de módulos para desarrollar un backend eficiente y escalable.

Estas son algunas de las librerías que usaremos para la implementación.

### 1. Framework: Express.js

Utilizamos Express.js como nuestro framework web, que proporciona una estructura mínima y flexible para construir aplicaciones web y APIs.

### 2. Gestión de paquetes: npm

Node.js viene con npm, el gestor de paquetes por defecto de Node.js, que nos permite instalar y administrar dependencias fácilmente.

### 3. Autenticación y autorización: Passport.js

Podríamos utilizar Passport.js para gestionar la autenticación de usuarios. Es un middleware extremadamente flexible que nos permite autenticar a los usuarios utilizando diferentes estrategias, como local, JWT, OAuth, etc.

### 4. Base de datos: MongoDB

MongoDB es una base de datos NoSQL que se integra bien con Node.js y es ideal para aplicaciones que requieren flexibilidad en el esquema de datos, como un sistema de comercio electrónico. Mongoose es una biblioteca de modelado de objetos MongoDB que simplifica la interacción con la base de datos.

### 5. Procesamiento de pagos: Stripe.js

Para gestionar los pagos, podríamos integrar Stripe.js, una API de pagos en línea que proporciona una forma sencilla y segura de aceptar pagos en nuestra aplicación.

### 6. Pruebas: Jest, Supertest

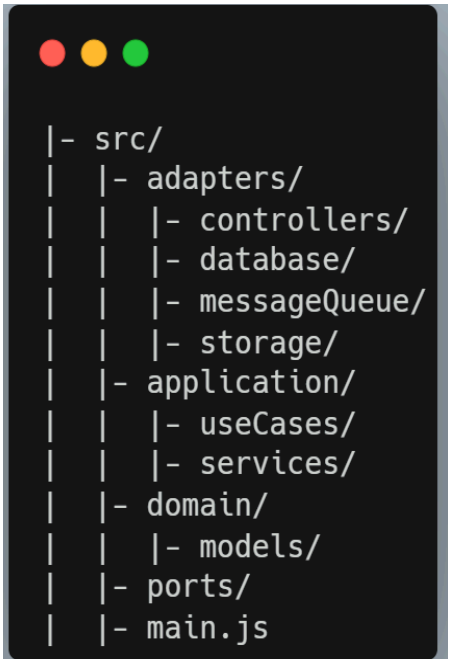
Para las pruebas, podríamos utilizar Jest como nuestro marco de pruebas y Supertest para realizar pruebas de integración en nuestras APIs.

### 7. Almacenamiento de imágenes: S3

S3 es un servicio de almacenamiento de objetos escalable y altamente disponible. Podríamos utilizar S3 para almacenar imágenes de productos, perfiles de usuario, etc.

## Estructura de carpetas

Para estructuras los archivos de nuestro proyecto utilizaremos la arquitectura hexagonal, Con esta estructura, mantenemos una clara separación entre la lógica de negocio, la infraestructura y los detalles de implementación. Además, podemos cambiar fácilmente los adaptadores sin afectar la lógica de la aplicación, lo que hace que nuestro sistema sea más flexible y fácil de mantener.




### Descripción de los directorios

1. adapters/: Contiene los adaptadores que se encargan de interactuar con tecnologías externas, como controladores HTTP, bases de datos, colas de mensajes y servicios de almacenamiento.
2. application/: Contiene los casos de uso de la aplicación y los servicios que implementan la lógica de negocio. Los casos de uso son independientes de la infraestructura y se comunican con los puertos para realizar operaciones.
3. domain/: Contiene los modelos de dominio de la aplicación, que representan las entidades y reglas de negocio. Estos modelos son independientes de la infraestructura y de los detalles de implementación.
4. ports/: Define los puertos que la aplicación expone para interactuar con el mundo exterior. Los puertos son interfaces que definen las operaciones que la aplicación puede realizar, pero no implementan la lógica concreta. Los adaptadores implementan estos puertos.

## Patrones de diseño

### 1. Patrón de Diseño Singleton

Podemos utilizar el patrón Singleton para garantizar que ciertas clases solo tengan una única instancia en toda la aplicación. Por ejemplo, podríamos aplicarlo a las clases de servicios que necesitan tener una única instancia en todo el ciclo de vida de la aplicación, como un servicio de autenticación o un servicio de conexión a la base de datos.

### 2. Patrón de Diseño Factory:

El patrón Factory nos permite encapsular la creación de objetos y ocultar los detalles de implementación. Podríamos aplicarlo para crear instancias de adaptadores de bases de datos, adaptadores de colas de mensajes, etc., de manera que podamos cambiar la implementación con facilidad si es necesario en el futuro.

### 3. Patrón de Diseño Repository:

El patrón Repository nos permite abstraer la lógica de acceso a los datos de la aplicación, proporcionando una interfaz común para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades de dominio. Esto nos permite desacoplar la lógica de negocio de los detalles de implementación de la persistencia de datos.

### 4. Patrón de Diseño Strategy:

El patrón Strategy nos permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Esto nos permite cambiar dinámicamente el comportamiento de un objeto en tiempo de ejecución. Podríamos aplicarlo, por ejemplo, para manejar diferentes estrategias de pago en nuestra aplicación de comercio electrónico.

### 5. Patrón de Diseño Observer:

El patrón Observer nos permite definir una dependencia uno a muchos entre objetos de manera que cuando un objeto cambia de estado, todos sus dependientes sean notificados y actualizados automáticamente. Esto podría ser útil, por ejemplo, para notificar a los clientes sobre cambios en el estado de sus pedidos.

### 6. Patrón de Diseño Dependency Injection (Inyección de Dependencias):

El patrón de Inyección de Dependencias nos permite desacoplar las dependencias entre las clases y facilita la prueba unitaria y la modularidad. En nuestro caso, podríamos utilizar un contenedor de inversión de control (IoC) para administrar las dependencias y realizar la inyección de dependencias en nuestras clases.

## Control de versiones

Escoger Git como sistema de control de versiones es una excelente elección para cualquier proyecto de desarrollo de software, incluido nuestro sistema de comercio electrónico. Git es ampliamente utilizado en la industria y ofrece una serie de beneficios, como control de versiones distribuido, ramificación fácil y funciones sencillas.

# Modelo de despliegue

## 1. Infraestructura en la Nube:

- Utilizaremos un proveedor de servicios en la nube como AWS, Google Cloud Platform (GCP) o Azure para alojar nuestra aplicación.
- Crearemos instancias de máquinas virtuales (VM) o contenedores para ejecutar nuestra aplicación. También aprovecharemos servicios gestionados como AWS Elastic Beanstalk, Google Kubernetes Engine (GKE) o Azure App Service para facilitar el despliegue y la gestión de la aplicación.

## 2. Orquestador de Contenedores:

- Utilizaremos Kubernetes como orquestador de contenedores para administrar y escalar nuestros contenedores de aplicación de manera eficiente.
- Desplegamos nuestra aplicación como contenedores Docker en Kubernetes para garantizar la portabilidad y la consistencia del entorno de ejecución.

## 3. Pipeline de CI/CD:

- Configuraremos un pipeline de CI/CD utilizando herramientas como GitHub Actions, Jenkins o GitLab CI para automatizar el proceso de construcción, pruebas y despliegue de nuestra aplicación.
- El pipeline se ejecutará automáticamente en cada push a la rama main (o master) y desplegará automáticamente la aplicación en nuestro entorno de producción después de que las pruebas hayan pasado con éxito.

## 4. Infraestructura como Código (IaC):

- Utilizaremos herramientas como Terraform o AWS CloudFormation para definir nuestra infraestructura como código, lo que nos permitirá gestionar y desplegar nuestra infraestructura de manera programática y reproducible.
- Definiremos todos los recursos de infraestructura necesarios, como instancias de máquinas virtuales, clústeres de Kubernetes, redes y permisos de seguridad, utilizando archivos de configuración que puedan versionar junto con nuestro código fuente.